LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-678165

MPMD Framework for Offloading Load Balance Computation

O. T. Pearce, T. G. Gamblin, B. R. de Supinski,
M. Schulz, N. M. Amato

October 13, 2015

International Parallel and Distributed Processing Symposium
Chicago, IL, United States
May 23, 2016 through May 27, 2016

Disclaimer

MPMD Framework for Offloading Load Balance Computation

Olga Pearce*, Todd Gamblin*, Bronis R. de Supinski*, Martin Schulz* and Nancy M. Amato†
*Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore CA 94550, USA
Email: {olga,tgamblin,bronis,schulzm}@llnl.gov
†Department of Computer Science and Engineering, Texas A&M University, College Station TX 77843–3112, USA
Email: {amato}@cse.tamu.edu

***Abstract*—In many parallel scientific simulations, work is assigned to processors by decomposing a spatial domain consisting of mesh cells, particles, or other elements. When work per element changes, simulations can use dynamic load balance algorithms to distribute work to processors evenly. Typical SPMD simulations wait while a load balance algorithm runs on all processors, but this algorithm can itself become a bottleneck.**

We propose a novel approach based on two key observations: (1) application state typically changes slowly in SPMD physics simulations, so work assignments computed in the past still produce good load balance in the future; (2) we can decouple the load balance algorithm so that it runs concurrently with the application and more efficiently on a smaller number of processors. We then apply the work assignment "late", once it has been computed. We call this approach *lazy load balancing*.

In this paper, we show that the rate of change in work distribution is slow for a Barnes-Hut benchmark and for ParaDiS, a dislocation dynamics simulation. We implement an MPMD framework to exploit this property to save resources by running a load balancing algorithm at higher parallel efficiency on a smaller number of processors. Using our framework, we explore the trade-offs of lazy load balancing and demonstrate performance improvements of up to 46%.

I. Introduction

Developers of high performance simulations must ensure that computational work is balanced among processes to maximize parallel efficiency. Current supercomputers comprise millions of processors, and as concurrency levels continue to increase, balancing work becomes increasingly difficult. Re-balancing work at runtime is necessary because many simulations have workloads that evolve dynamically. Some codes use mesh cells, particles, and other logical elements to represent their domains, but the computational work per element can change over time, causing the application to become imbalanced.

The cost of imbalance increases with scale. Most scientific simulations use an SPMD parallel programming model, and underloaded processes must wait for overloaded ones to complete before continuing. The larger the application run, the more processing resources are wasted by a single slow process. Thus, we must fix even small imbalances at scale. Moreover, in the strong-scaling limit, balancing work becomes increasingly difficult as the available parallelism becomes more coarse-grained with respect to the number of processes.

Many large-scale parallel applications use load balance algorithms to redistribute work evenly. Depending on the application, a fast, local load balance algorithm may be suitable. However, graph partitioners are typically employed for the best balance, efficient communication optimizations, and for work assignment to be aware of locality within the simulated physical domain [8], [21]. Graph partitioners are computationally intensive, require sophisticated parallelization, and typically exhibit worse strong scaling than the simulation itself. This makes them too expensive at scale.

In this paper, we describe *lazy load balancing*, a new approach that allows efficient large-scale applications to use load balance algorithms that distribute load well and optimize communication, even if they have high latency and scale poorly. We decouple the load balance algorithm from the application and run it on separate processors. This removes load balance computation from the application's critical path, and it allows the application and the load balancer to run at their most efficient respective scales. In this MPMD configuration, work is reassigned *lazily* in the application program as assignments become available from the load balancer.

Lazy work assignment must handle application state changes that occur while the load balance algorithm runs. These changes can impact the work distribution, and we call this change *drift*. Fortunately, state changes slowly in most applications, so a work assignment computed from stale state typically continues to be a good assignment for many time steps. Lazy load balancing guarantees a correct application state after work reassignment, even with application drift.

Our contributions include:

- A lazy load balancing framework;
- An empirical evaluation of drift metrics for two applications, Barnes-Hut and ParaDiS;
- An analytical model that predicts the right size for a load balance partition from application characteristics.

We show that our approach can improve performance by up to 46%, even for applications with substantial drift.

Section II describes related work. Section III outlines our lazy load balancing approach. Section IV describes our MPMD framework for offloading the load balance algorithm. Section V introduces our model for deciding if lazy load balancing will be beneficial. Section VI outlines the application properties and drift metrics that make lazy load balancing feasible. Section VI-A describes the applications that we balance along with their drift metrics. Section VII shows our results.

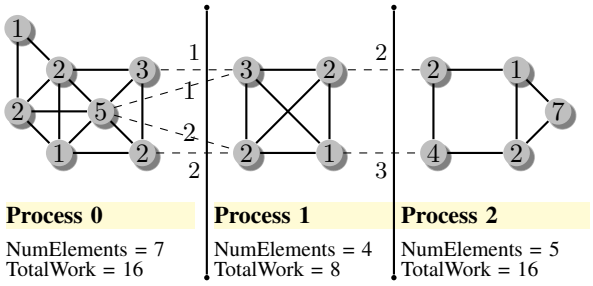


Fig. 1: Distributed Application Element Graph

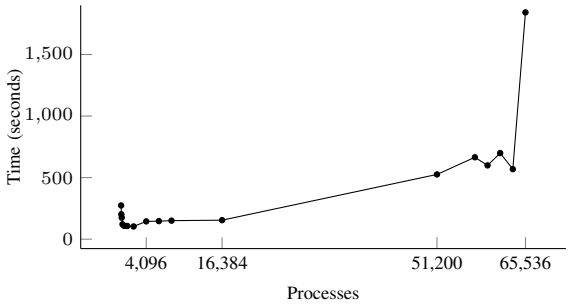


Fig. 2: Graph Partitioner Runtime. BGQ, 265K vertices

II. Background and Related Work

Most SPMD physics codes divide the simulated domain into logical elements, which are assigned to processors in the parallel machine. Often, the work per element varies over time, so evenly dividing elements among processors does not guarantee an even distribution of work. Figure 1 shows an application domain with elements divided among three processes. The domain is represented as a graph with elements as nodes weighted by their computational work. Communication dependencies among elements are shown as edges. Dashed edges represent (weighted) interprocess communication.

Periodically, simulations use dynamic load balance algorithms to reassign work to processors. An accurate assignment must solve the balanced graph partition problem, which is NP complete [13]. In practice, many applications employ local algorithms to balance work only among neighboring processes. This does not balance work or minimize communication as well as graph partitioning [17]. Thus, applications that need more balance use a graph partitioner. While graph partitioning is well studied and many heuristics exist [8], [21], *parallel* graph partitioning algorithms scale poorly. Figure 2 shows strong scaling performance of a parallel graph partitioner running on an IBM Blue Gene/Q for 32 to 65,536 processors. Peak efficiency occurs with 2,048 processors, after which, runtime increases. On 65,536 processes, the algorithm spends all of its time in communication, and the runtime skyrockets. Poor scalability leads many applications to use partitioning only offline. Others run a partitioner inefficiently at large scale, as memory constraints can prohibit running on fewer nodes.

Recent work has partially addressed the scalability of parallel graph partitioners by assigning coordinates to graph components in a lattice-based multilevel embedding [14]. This solution can work for some applications but scalable solutions to the graph partitioning problem are challenging because of the trade-off between computing high-quality partitions and limiting the added communication. Our approach enables high-accuracy graph-partitioning at runtime by addressing:

1) **Scalability:** We designed a framework to tailor the process count used by the load balance algorithm; and
2) **Memory Constraints:** We use sampling to reduce memory requirements of the load balance algorithm.

Many early load balancing methods on smaller parallel machines ran on a single processor [15]. Their scale was effectively *decoupled* from that of the application. Our work builds on this by optimizing decoupled *parallel* load balancers.

We build on the concurrent programming language notions of futures, promises, and lazy evaluation. Lazy evaluation [26] delays the evaluation of an expression until its value is needed. We delay application rebalancing until directions are known.

Charm++ [4] runs the load balance algorithm asynchronously to the application. However, Charm++ has a parallel object model that considers coarse grained objects and may be impractical for tightly coupled parallel applications. Charm++ does not have a predictive model of the full load balancing system, or the ability to right size the resources for the balancing algorithm. We present a solution that preserves the computational model in the application and does not require modifications to existing MPI applications.

Overlapping computation and communication is a well-known technique to optimize parallel performance [7] for different algorithms [11] and architectures [27]. Resources have been dedicated to collective operations [19], I/O [25], checkpointing [20], and tool services [1]. We overlap application and load balance computations.

III. Lazy Load Balancing Approach

Figure 3(a) shows the main components of the traditional approach to load balancing an application. The main steps are:

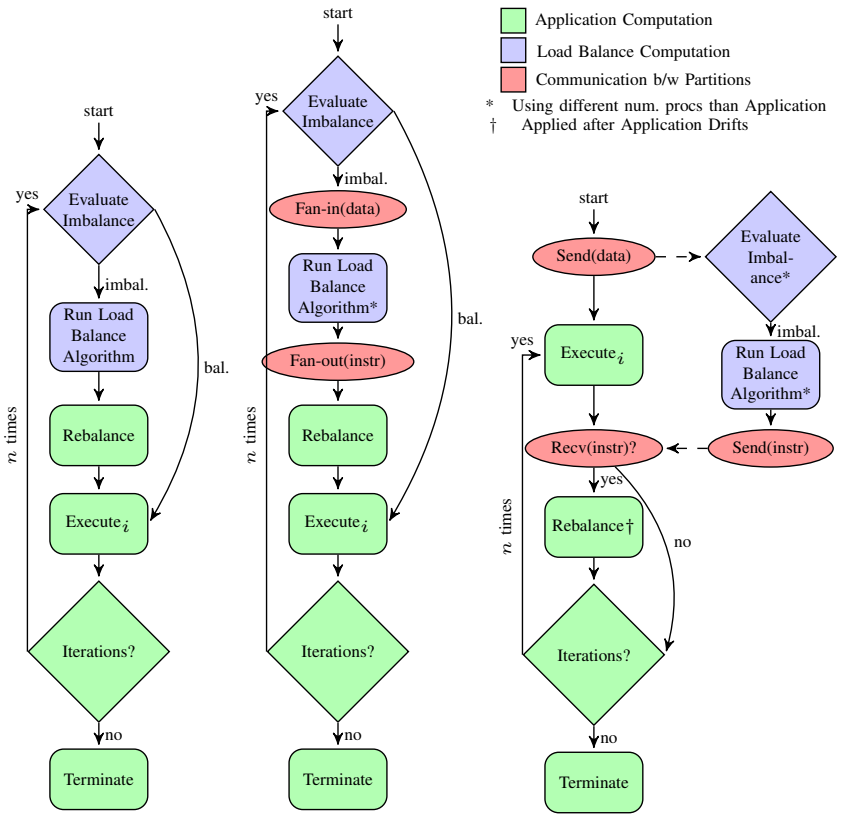
1) *Evaluate Imbalance*: Decide whether to correct load imbalance at this point in execution;
2) *Run Load Balance Algorithm*: Use a load balance method to compute directions on how to rebalance;
3) *Rebalance* the application if needed.

Figure 3(a) and Algorithm 1 demonstrate how these steps are typically performed while the application's primary computation is paused. As discussed, this approach is not well suited to using a graph partitioner; it may leave thousands of processes waiting while the partitioner runs at low efficiency.

A. Decoupling the Load Balance Algorithm

Load balancers are distinct from applications, and the total work of load balancing is nearly always smaller than the application's computation. Using the same number of processes as the application, the load balancer has much less available parallelism because the granularity of work assignment is typically much coarser than the resolution of the simulation.

We *decouple* the resources used by the load balance algorithm from those used by the application. We move the data



(a) Inline LB (b) Decoupled LB (c) Lazy LB

Algorithm 1 Inline LB $(P_{App} == P_{LB})$

1: **if** procID $p_i \in P_{App}$ **then**
2: **Pause** computation, Evaluate Imbalance
3: Run Load Balance Algorithm
4: Rebalance
5: **Proceed** with balanced computation

Algorithm 2 DecoupledLB $(P_{LB} \subset P_{App})$

1: **if** procID $p_i \in P_{App}$ **then**
2: **Pause** computation, Evaluate Imbalance
3: Send input to corresponding P_{LB_j}
4: **if** procID $p_j \in P_{LB}$ **then**
5: Receive input from all corresp. P_{App_i}
6: Run Load Balance Algorithm
7: Send output to all corresponding P_{App_i}
8: **if** procID $p_i \in P_{App}$ **then**
9: Receive output from corresponding P_{LB_j}
10: Rebalance
11: **Proceed** with balanced computation

Algorithm 3 LazyLB $(P_{App} \cap P_{LB} == \emptyset)$

1: **if** procID $p_i \in P_{App}$ **then**
2: Send input to corresponding P_{LB_k}
3: **Proceed** with imbalanced computation
4: **if** procID $p_k \in P_{LB}$ **then**
5: Receive input from all corresp. P_{App_i}
6: Evaluate Imbalance, Run LB Algorithm
7: Sent output to all corresponding P_{App_i}
8: **if** procID $p_i \in P_{App}$ **then**
9: Receive output from corresponding P_{LB_k}
10: **Pause** computation
11: Rebalance
12: **Proceed** with balanced computation

Fig. 3: Load Balancing Configurations: Inline, Decoupled, and Lazy

to be partitioned onto a set of processes different from those used by the application, allowing less scalable algorithms to run more efficiently on fewer processes. While this approach may seem obvious, decoupling is rarely if ever used in practice. Instead, applications use all processes, or they use a centralized approach with obvious scaling limitations. Our approach enables consolidation to fewer processes, making it possible to run load balance algorithms more efficiently.

Figure 3(b) and Algorithm 2 illustrate our decoupled approach. First, the application pauses and sends its state to the load balance processes (lines 2-3). Load balance processes receive the application state (line 5), run in parallel (line 6), and send an assignment instructions back to the application (line 7). The application receives the instructions (line 9), rebalances (line 10), and proceeds with the computation in a balanced state (line 11). This approach optimizes the resources used by the load balance algorithm but leaves resources unused while the application waits for the load balancer.

B. Lazy Load Balance Algorithm

To avoid pausing the application computation while computing a load balance assignment, we can *offload* the load balance computation to a separate *balancing partition*. This allows more *concurrency*, as it overlaps application computation and load balance computation. Assignments are applied by the application *lazily* as they become available.

Algorithm 4 Application Interface

1: **for** timesteps **do**
2: Execute application iteration
3: Provide Distributed Application Element Graph to Lazy LB
4: **if** instructed to rebalance **then**
5: Reassign work as directed by MPMD Framework

Figure 3(c) and Alg. 3 demonstrate lazy load balancing. First, the application sends its state to the load balance processes (line 2). The computation proceeds imbalanced (line 3). The load balance processes receive the application state (line 5), run the load balance algorithm in parallel (line 6), and send the instructions back to the application processes (line 7). The application then receives instructions, pauses, rebalances (lines 9-11), and proceeds in a balanced state (line 12).

A key difference with Alg. 2 is that when the application receives load balance instructions (line 9), it has progressed from its prior state. We assume that the application can only be rebalanced at timestep boundaries. The load balance algorithm uses a snapshot s_0 of the application at timestep t_0 to compute how to rebalance the application; we refer to this decision as d_0. Assume that d_0 results in a balanced application state. During load balancing, the application advances k timesteps. Applied to s_k, d_0 may result in an imbalanced state.

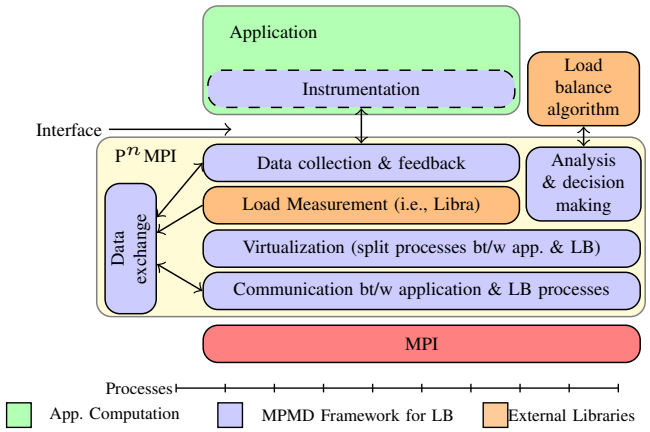


Fig. 4: Lazy Load Balancing Infrastructure Components

IV. MPMD Framework

We have developed a framework for decoupling the resources used by the load balance algorithm from the application resources without changes to the actual computation in the application. Our framework enables offloading the load balance algorithm to separate processes, allowing it to execute asynchronously, without pausing the application.

The requirements on the application are the same as for adding an external load balance algorithm, namely to provide the input to the load balance algorithm, and to be able to act on the rebalancing directions received from it. We describe the application interface in Algorithm 4. The application provides a distributed application element graph (described in Section II, Figure 1) to our framework, just as it would provide it to a stand-alone load balance algorithm. Our framework runs the load balance algorithm in its own partition, and handles the data movement between the application and the load balance algorithm partition. When directed to rebalance, the application is responsible for moving work units to processes indicated by the load balance algorithm.

We achieve this separation by *virtualizing* MPI and executing the application on a different, potentially smaller, set of processes than the job allocation, as described in Section IV-A. We describe our *asynchronous* communication protocol between the application and balancing processes in Sec.IV-B.

A. Reserving Resources for Load Balancing

Without loss of generality, we focus our implementation on applications built on top of the Message Passing Interface (MPI) [16], currently the de-facto standard for large scale scientific applications. Our approach requires separate processors for the load balance algorithm. We implement a portable mechanism in MPI to allocate the processes in an MPMD fashion *without modifying the application*. Our framework virtualizes MPI_COMM_WORLD using the `PMPI` interface, and it allocates separate communicators for the load balancer and the application, within the same job partition.

We use P^{N}MPI [22] to integrate modules in our framework. P^{N}MPI is a framework that stacks independent tools built using the MPI profiling interface [12]. We use P^{N}MPI to integrate the tools across different process partitions: the measurement tools, the interaction with the application, and our new asynchronous component. Our tools add negligible overhead to the application runtime.

The virtualization module intercepts all application communication, and transparently replaces MPI_COMM_WORLD with a smaller communicator, APP_COMM and with that provides the illusion for the application that it is executed on a smaller set of resources. We can choose arbitrary processes within MPI_COMM_WORLD to run the load balance algorithm; selecting an optimal physical placement among the application processes depends on the application and the host platform and is beyond the scope of this work. With processes reserved for the load balance algorithm, the application proceeds, creating its own communicators from the virtualized, smaller MPI_COMM_WORLD. The load balance algorithm executes asynchronously on a separate communicator, LB_COMM, where MPI_COMM_WORLD = APP_COMM $\cup$ LB_COMM.

Figure 4 shows the interaction of the P^{N}MPI modules in our framework with the application and libraries. The virtualization module splits the processes between the two communicators. Data collection is performed on APP_COMM, which allows the use of additional data collection modules like Libra [10]. All load balance calculations are run in LB_COMM, including external load balance libraries like Zoltan [8]. LB_COMM processes also determine whether to send the application rebalance instructions.

B. Asynchronous Interaction Protocol between Application and Load Balancing Processes

Our tools allow a load balancer to execute asynchronously, outside the application. Figures 6 and 7 show the communication protocol used by the application and the load balance processes. Figure 5 shows the shorthand for the valid state transitions, which indicate when the load representation (graph) needs to be sent and when the application should rebalance. The application sends information about its state and continues to run while the load balance algorithm computes. The execution within the load balancing processes can be synchronous, allowing us to call MPI libraries (i.e., partitioners), directly, without impacting application execution. The application is sent the rebalancing instructions only when they have been computed. The application can then apply this decision at the next stopping point (i.e., between time steps).

If the application is at a stopping point and has not received rebalancing instructions, it can:

- Always wait for load balancing directions (inline);
- Never wait for balancing directions (fully asynchronous);
- Continue for up to a given number of time steps and then wait (middle ground).

The above decision can be made at runtime for each stopping point and should be based on a function of how quickly the application load balance deteriorates. The remainder of this paper evaluates the fully asynchronous mode, to show the performance benefits of lazy load balancing.

G	Send Graph
R	Rebalance

G	R
0	0
1	0
1	1

Fig. 5: Valid State Transitions

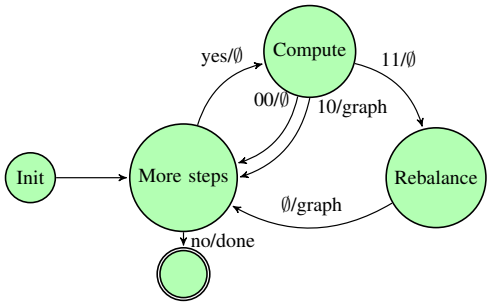


Fig. 6: Application State Machine. Transitions Show Input (from Load Balance Partition) / Output (from Application to Load Balance Partition)

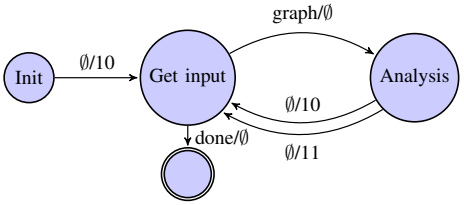


Fig. 7: Load Balance Partition State Machine. Transitions Show Input (from Application) / Output (from Load Balance Partition to Application)

TABLE I: Break Down of Execution Time for Standard, Decoupled, and Lazy Approaches to Load Balancing

p	Total number of processes	t	Number of timesteps	$f(n,w)$	Runtime of a balanced timestep, on n procs
n	Processes used by Application	u	Num. of unbal. timesteps	$g(m,w')$	LB runtime on m processes
m	Procs used by Decoupled LB	t'	N.steps until imbal.threshold	$h(m,p,w')$	Comm. bt/w m Decoupled LB & p App. procs
m'	Processes used by Lazy LB	t_a	Actual steps bt/w rebal.	$h'(m',n,w')$	Comm. bt/w m' Lazy LB and n App. procs
w	Work in the Application	α	Initial Application imbalance	$r(n,w)$	App. redistribution time, run on n processes
w'	Work in the LB algorithm	i	Rate of change in imbalance	tol	Imbalance threshold

	Standard	Decoupled	Lazy
Processes	$p = m = n$	$m \leq n = p$	$m' + n = p$
Lazy Steps	$u = 0$	$u = 0$	$u > 0$, $u = \frac{g(m,w')+h'(m',n,w')}{\alpha\ f(n,w)}$
Actual Steps bt/w Rebalancing	$1 \leq t_a \leq t'$	$1 \leq t_a \leq t'$	$u \leq t_a \leq t'$
LBTime	$g(p,w)$	$g(m,w') + h(m,p,w')$	$g(m',w') + h'(m',n,w')$
Application Time	$t\ f(p,w)$	$t\ f(p,w)$	$t\ f(n,w) + u\alpha\ f(n,w)$
Total Time	$g(p,w') + t\ f(p,w)$	$g(m,w') + h(m,p,w') + t\ f(p,w)$	$t\ f(n,w) + u\alpha\ f(n,w)$

V. Resource Allocation Model

To minimize overall application runtime by selecting the best load balancing configuration, we develop a performance model that provides a quantitative basis for deciding how to allocate the available resources in the system. The cost model captures the performance characteristics of the standard, decoupled, and lazy load balancing configurations. To simplify the explanation, we assume that the load balance algorithm is able to balance the application fully. As application performance prediction is beyond the scope of this work, we use the cost and benefit analysis at a given point in the simulation from our previous work to decide whether a rebalancing would be beneficial [18].

Figure 8 illustrates how the different load balancing configurations result in different resource usage. Table I summarizes the variables used in the performance model. The height of the block indicates the number of processes (resources); the length of the block indicates the runtime. We consider application computation, load balance algorithm computation, and communication overhead. The lazy approach optimizes resource usage by running the load balance algorithm on fewer processes, and by overlapping the load balance algorithm with the main computation. Running the load balance algorithm asynchronously means it can run continuously, correcting the imbalance with a higher frequency than otherwise might be affordable. Figure 8(d) demonstrates how the application can continue sending its state to the load balance processes, and the load balance algorithm can continually compute the rebalancing directions. The frequency of the rebalancing should be based on the amount of *drift* that an application can tolerate, as discussed it in Section VI vs. the number of resources a user is willing to dedicate to load balancing and with that the frequency in which new work assignments can be produced.

Model Input:

- p – Total number of processes;
- α – Initial application imbalance;
- t – The smaller of the number of time steps that the simulation takes, or the number of steps the simulation takes prior to *drift* becoming too large;
- Runtime of the application, load balance algorithm, and communication overhead, as defined in next subsections.

Model Output:

- Choice of standard, decoupled, or lazy configuration;
- m – Number of processes for the load balance algorithm;
- n – Number of processes for the application.

Performance modeling of specific load balance and physical simulation algorithms are research topics in their own right and are outside the scope of this paper. Our model uses curve fitting for the load balance algorithm and the application runtime to determine how to allocate the resources in the system.

A. Modeling the Application

The computation time of an application, $f(n,w)$, is a function of the number of processes and the amount of work in the application, and we model it with a curve fit. The application

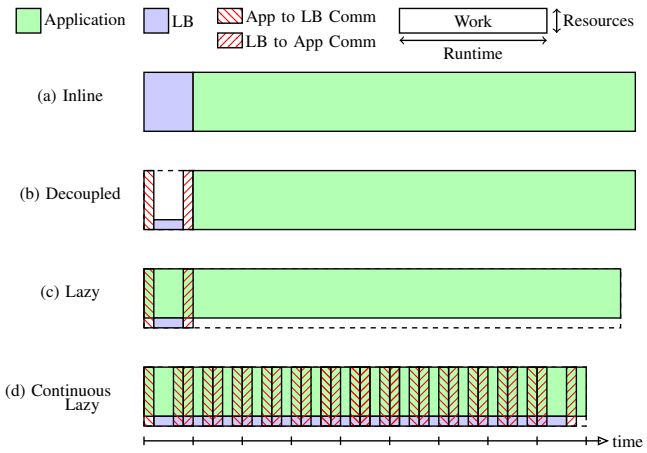


Fig. 8: Resource Diagram of Load Balancing Configurations

uses p processes in the inline and decoupled configurations, and $n < p$ processes in the lazy configuration.

At the beginning of the execution, the application has an initial imbalance α. Because the standard and decoupled configurations pause the application execution while the load balance algorithm computes, the application is redistributed immediately and computes all t timesteps in a balanced state. In the lazy configuration, the application continues running imbalanced for u steps, while the offloaded load balance algorithm computes the directions; we discuss how we determine u in the next subsection. Once the rebalancing directions are received, the application is redistributed and completes the remaining $t - u$ steps in a balanced state. The application execution time therefore depends on whether the timesteps are executed prior to or after redistribution, and how many processes the application uses.

The application provides t, the number of time steps it will execute. For simplicity, we assume that the *drift* metric i, rate of change in imbalance, is constant. We estimate i at runtime or accept it as input from the application. Using the rate of change, we compute how many steps t' the application can execute before it needs to be rebalanced again because it reaches an imbalance threshold tol:

$$t' = \frac{\log(tol \times f(n,w)) - \log(f(n,w))}{\log(1+i)} \tag{1}$$

We use t' as the largest number of steps after which the application needs to be rebalanced. Additionally, we empirically model the cost of application redistribution, r, which is a function of the data needed by the application and the number of processes used by the application.

B. Modeling the Load Balance Algorithm

The computation time of a load balance algorithm, $g(m, w')$, is a function of the amount of work and the number of processes used. The amount of work in the load balance algorithm, w', is usually smaller than the amount of work in the application itself, w.

The performance of each load balance algorithm is modeled with curve fitting, and the algorithm will use p processes in the standard configuration, $m \leq p$ processes in the decoupled configuration, and $m' < p$ processes in the lazy configuration.

In the decoupled and lazy configurations, the application processes first send relevant information to the load balance processes. Once the load balance directions are computed, they are sent back to the application. We refer to this overhead, $h(m, p, w')$ and $h'(m', n, w')$, as communication overhead for decoupled and lazy configurations. In our implementation, we use lazy communication between the application processes and the load balance processes. However, for simplicity we model this communication as if it cannot be overlapped with the computation. The runtime of gather/scatter operations depends on the number of the application processes and load balance processes, and the amount of data sent.

In the lazy configuration, the application proceeds in an imbalanced state for u steps, while the load balance information is gathered, the load balance algorithm executes, and the directions are scattered back to the application. We calculate u as a fraction of the time until the directions are available, and the length of the imbalanced timestep:

$$u = \frac{g(m, w') + h'(m', n, w')}{\alpha\, f(n, w)} \tag{2}$$

The extra time that the application runs because of the delay in rebalancing becomes part of the overhead of the lazy load balancing configuration.

C. Modeling Overall Runtime

The total time for the standard and decoupled configurations is a sum of load balance algorithm time and application time, including the gather/scatter overhead in the decoupled configuration. The total time for the lazy configuration only includes the application time since the load balance algorithm time is overlapped; however, the application runs using fewer processes because some of the resources were reserved for the load balance algorithm, so the application runtime may be longer. Additionally, as discussed in Section V-B, the application runs in an imbalanced state for u steps, which increases the total time as well.

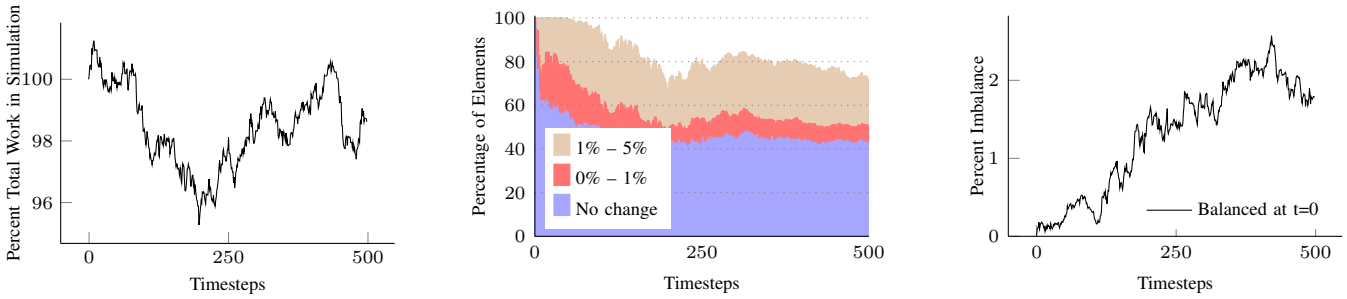
The decoupled configuration is a generalization of the standard configuration when $m = n = p$, $h(m, p, w') = 0$, so we only discuss how to choose between the decoupled and lazy configurations. Given p, the model chooses m, n, and t_a s.t.:

$$min \begin{cases} g(m, w') + h(m, p, w') + t\ f(p, w) \\ t\ f(n, w) + u\alpha\ f(n, w) \end{cases} \tag{3}$$

We discuss an instantiation of our model for our test applications and load balance algorithm in Section VII-B.

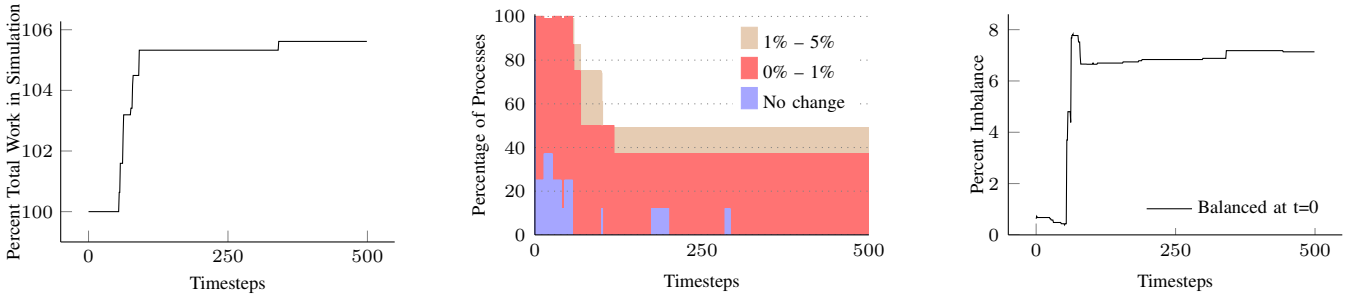
VI. Application Drift

A potential problem with lazy load balancing is that application state changes over time. Work per element may change, as may the number of elements. We call this change *drift*. However, the work distribution in most parallel SPMD applications changes slowly as applications carefully choose timestep length or use adaptive time-stepping for computational stability [23]. Often, a balanced assignment computed



(a) Total Work Over Simulated Time (b) Work per Element Over Simulated Time (c) Imbalance Over Simulated Time

Fig. 9: Application Drift in Barnes-Hut, 26M Interactions, 8192 Processes



(a) Total Work Over Simulated Time (b) Work per Process Over Simulated Time (c) Imbalance Over Simulated Time

Fig. 10: Application Drift in ParaDiS, 16M Interactions, 2048 Processes

from a past application state is a good approximation of a balanced assignment for the current state, so we can compute the assignment asynchronously and apply it *lazily* when available.

Given an assignment $A{:}V{\to}P$ for a past time step's elements V, we compute a *drifted* assignment $A'{:}V'{\to}P$ for the current time step's elements V' by determining the relationship between V and V'. There are three possibilities:

$$A'(v) = \begin{cases} A(v) & v \in V \cap V' \\ C(v) & v \in V' \setminus V \\ undefined & v \in V \setminus V' \end{cases}$$

In the first case, v is in both the old and the new application state, and we reuse its assignment from A. This case will cover most of the elements, and it is the only case for most N-body simulations and for unstructured mesh applications with a static number of cells in the mesh, because in these applications $V = V'$. In the second case, v represents an application element or task that has been created since the past state, and we must construct a new function $C : V'\setminus V \to P$ to assign it. In the third case, v represents an element or task that no longer exists, and we can ignore its prior assignment. The second and third cases are typical of adaptive mesh refinement (AMR) algorithms and other methods in which elements may be created based on the physics. Computing a good function for C is application-dependent and we leave its description to later sections. We next define what it means for an assignment to be *valid* and *efficient*.

Assignment Validity: For A' to be *valid* for the new application state it must be defined for all of V'. Trivially, A' is defined for $(V \cap V') \cup (V' \setminus V)$, which equals V'.

Assignment Efficiency: An assignment is *efficient* if it minimizes the deviation in process load, minimizing load imbalance. We define the *imbalance* of an assignment $I(A)$ as the scaled maximum load on any processor minus the average:

$$I(A : V \to P) = \frac{\max_i(W(V_i)) - \frac{1}{|P|}\sum_i W(V_i)}{\frac{1}{|P|}\sum_i W(V_i)} \quad (4)$$

In a bulk synchronous application, the extra load corresponds to the performance degradation of any overloaded processors.

A. Empirical Evaluation of Drift Metrics

To show the effect that drift has on load balance, we evaluate the difference between the *drifted* assignment A' and the original assignment A. Here, we look at N-body methods, an important class of simulations critical for research in fields such as molecular dynamics, astrophysics, and material science. N-body methods simulate the evolution of systems of particles (or bodies). Each particle may exert a force on any other. The simulation progresses by computing force interactions among particles, then updating the particles to reflect the force's effect. Force computations typically consume the bulk of the execution time, yet this computation can exhibit significant load imbalance, a major performance problem.

We measure drift for two N-body applications: a Barnes-Hut algorithm and a dislocation dynamics application, ParaDiS. We ran each for 500 timesteps, enough for several load balance computations. The difference between A' and A depends on the number of timesteps application has taken between them.

Barnes-Hut [3] is a classic N-body algorithm that uses an octree to compute approximately the force that the N particles

in the system exert on each other (e.g., through gravity). The N leaves of the octree are the individual particles, while the internal nodes summarize information about the particles contained in the subtree (i.e., combined mass and center of gravity). Particles that interact with other particles in nearby cells are computed directly, but for interactions with cells that are sufficiently far away, only one force computation is performed for their cells. Our distributed version of Barnes-Hut is based on a shared memory implementation from the Lonestar suite in Galois [6]. Our version is written in C++ with MPI for interprocess communication.

Figure 9 shows the drift metrics for Barnes-Hut. In Barnes-Hut, the number of particles remains the same throughout the simulation, but the interactions computed per particle can change as particles move because the simulation only computes gravitational interactions within a cutoff radius. The total work, summed over all particles in the simulation, fluctuates over a range of about 5% of the total initial work over time (Figure 9(a), total initial work is 100%). The work per particle in the simulation changes over time (Figure 9(b)); approximately half the particles have the same amount of work throughout the execution, but the other half's work changes up to 5% per time step. The imbalance that results from this change on each successive step (assuming that we use a completely balanced assignment from time step zero) grows only slightly (Figure 9(c)), despite the changes in work per element in the simulation. Thus, the assignment computed at step 0 is still an efficient assignment for subsequent steps.

ParaDiS [2], [5] is a large-scale dislocation dynamics simulation used to study the fundamental mechanisms of plasticity. The simulation includes $O(N)$ calculation of forces, the equations of motion, time integration, adaptive mesh refinement, the treatment of dislocation core reactions and the dynamic distribution of data and work on parallel computers. ParaDiS integrates all of the above algorithms to understand their behavior in concert and to evaluate the overall numerical performance of dislocation dynamics simulations and their ability to accumulate percent of plastic strain. ParaDiS is written in C/C++ with MPI for interprocess communication. It computes the short-range forces directly and uses multipole expansion [24] for long-range force computation.

Figure 10 shows drift metrics for ParaDiS. In ParaDiS, elements can be created or destroyed as the simulation continues, thus the total work in the simulation can grow much more rapidly than in Barnes-Hut (Figure 10(a), total initial work is 100%). Nearly all processors experience a change in workload on every time step (Figure 10(b)), and the changes grow as the simulation continues. The change in imbalance (Figure 10(c)) shows that the largest change in imbalance comes from the creation of elements, but that the use of a drifted assignment still results in an imbalance of at most 7%.

B. Load Balance Algorithm for N-Body Simulations

We use a precise load balance algorithm for N-body simulations from our previous work [17], which corrects load imbalance in a single step, and achieves better load balance than the available application-specific solutions. The algorithm consists of the following steps:

1) **Select work units with sampling.** Sample performed force computations (interactions); use samples to divide interactions into subsets, or *work units*. This is a coarsening step on the domain decomposition.
2) **Construct model.** Use work units, proximity info.
3) **Partition model.** Assign work units to p processes by partitioning the work units into p groups.

The load balance algorithm generates a hypergraph by extracting the particle interactions from the application. To keep the hypergraph small, the algorithm uses adaptive sampling to choose representative interactions for sets of interactions. The algorithm partitions the hypergraph, assigning the representative interactions to processes. The algorithm uses the hypergraph partitioner from Zoltan [9]. By assigning all of the interactions in a set to a process along with the representative, the algorithm assigns the interactions explicitly while preserving the locality.

Work unit persistence should be discussed in the context of both the application and the load balance algorithm. Although new particles cannot be created in Barnes-Hut, the amount of work the simulation performs changes as particles move in and out of interaction range. ParaDiS, on the other hand, creates and removes particles throughout its execution; the amount of work in ParaDiS varies as well. Because the load balancing algorithm we use makes the assignment based on samples rather than all particles, we are able to use the same mechanisms to establish a mapping of all current work units to the selected samples without the need to rely on complete particle mappings, ensuring the correctness of the simulation.

VII. Performance Evaluation

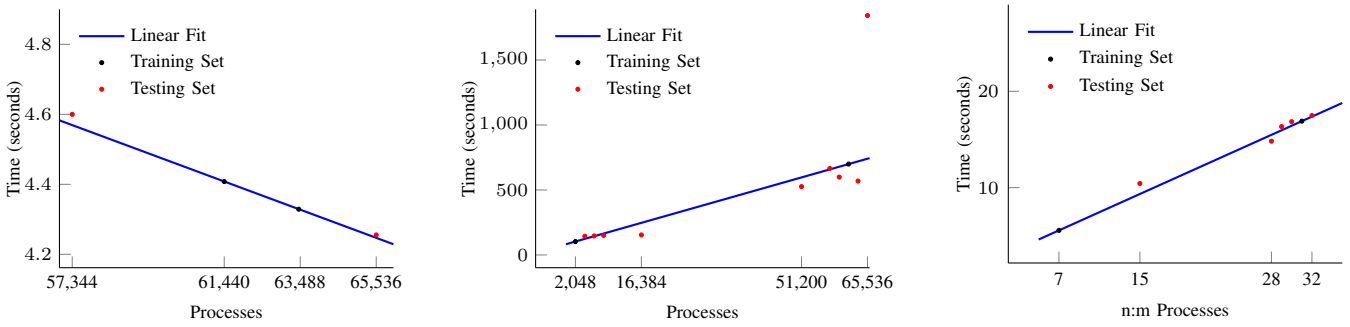
For our ParaDiS experiments, we use a Linux cluster with nodes consisting of two 2.8 GHz Hex-core Intel Xeon EP X5660 processors, twelve cores per node. All nodes are connected by QDR Infiniband. We use GCC 4.4.7 and MVAPICH v0.99 on top of CHAOS, an HPC variant of RedHat Enterprise Linux (RHEL), running at Linux kernel v2.6.32.

For Barnes-Hut experiments, we use an IBM Blue Gene/Q system, a tightly coupled MPP system that contains PowerPC based compute nodes with 16 cores (64 hardware threads) each. Nodes are connected by five dimensional torus network and run a simplified compute node OS, the CNK. We use GCC 4.4.6 and IBM's MPICH2-based MPI implementation.

A. Overhead of Lazy Load Balancing

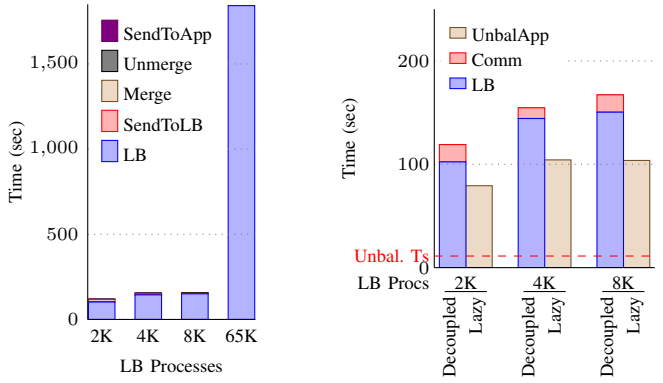
We show strong scaling for all comparisons. Weak scaling studies of N-body computations are difficult to construct accurately due to variability in the particle density during scaling. Showing strong scaling ensures a fair comparison.

Figure 11(a) shows the costs of decoupling the load balance algorithm as a function of the resources provided to the load balance algorithm. These costs include sending the data to the load balancing processes, merging the data from several application processes on a single load balancing process,



(a) Application Timestep Length (b) Load Balance Algorithm (c) Communication Parameters

Fig. 12: Resource Allocation Model Parameters Obtained via Curve Fitting



(a) Decoupling Overhead (b) Overhead of Configurations

Fig. 11: Lazy Load Balancing Overhead. BGQ, Barnes-Hut, 306.5M Interactions, 65,536 Application Processes

running the load balance algorithm in parallel on load balancing processes, unmerging, and sending the load balancing instructions back to corresponding application processes. The communication overhead imposed by the decoupling is proportional to the $n:m$ ratio of application processes to load balancing processes. When there is high fan-in, the load balancing processes have a larger serial communication cost. The time required to run the load balance algorithm at the same scale as the application is shown for comparison; we discuss the scaling of the balance algorithm in the next section.

Figure 11(b) shows the overhead of the decoupled and lazy load balancing configurations as a function of the resources provided to the load balance algorithm. In the decoupled configuration, overhead includes the load balance algorithm time and the communication overhead. In the lazy configuration, the overhead includes the time lost due to running the application in an imbalanced state while the load balancing directions are computed. Unbalanced timestep length is shown for scale.

B. Model Validation

Given the total number of processes available, we use our model to determine the load balancing configuration (no load balancing, inline, decoupled, or lazy) that will result in the shortest overall runtime, along with the number of processes to use to run the application and the load balancing algorithm.

TABLE II: Resource Allocation Model Parameters

p	Total available processes	65,536
α	Initial application imbalance	From App.
i	Rate of change in imbalance	From App.
t	Number of timesteps left	From App.
$f(n,w)$	Runtime of bal.timestep on n procs	Curve fit
$g(m,w')$	LB runtime on m processes	Curve fit
$h(m,p,w')$	Comm. bt/w m LB & p App. procs	Curve fit

Table II summarizes the model parameters. We use a training set of runs on a similar scale to model the load balance algorithm, application timestep, and communication overhead. Figure 12 shows a Barnes-Hut problem with 306.5M interactions on BGQ. We fit a curve to the training set measurements (shown as black dots); validation measurements are shown as red dots. We use least squares fit to model the parameters in our model. We model the load balance algorithm and the application timestep as a function of processes. The communication overhead, however, is closely related to the ratio between the number of application processes and the number of load balancing processes, so we model it as a function of the ratio (Fig.12(c)). At this scale, the application is still able to take advantage of using more processes, as demonstrated by the decreasing timestep length (Fig.12(a)). The graph partitioner becomes slower with additional resources, because the partitioned graph is small (265K vertices). At this scale, there is little work per process and high serialized overhead that increases with the number of processes (Fig.12(b)). This inefficiency is one of the motivations for our work: by decoupling the load balance algorithm we can use a graph partitioner when it would be too expensive to run at full scale.

Figure 13 (bottom) shows the total runtime of all approaches, including no load balancing at all. At top, we show the percent runtime improvement achieved using decoupled and lazy approaches. To simplify the comparison, a single load balancing step was performed for each approach. For *lazy* and *decoupled*, only the runtimes with the best performing parameters (number of processes used by application and load balance algorithm) are shown. We highlight the configuration selected by the model. In all but one case, the model accurately selects the best performing configuration. In the case that the

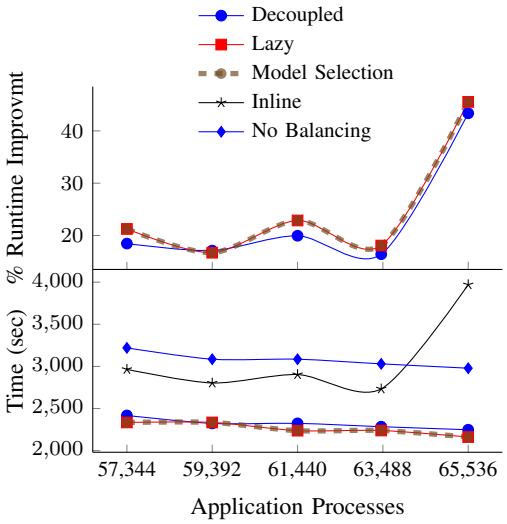


Fig. 13: Runtime and Improvement Over Inline LB

model selects the second best configuration, the difference in runtime between the first and second best configuration is small, so the error results in little performance loss.

Overall, the decoupled and lazy configurations are able to reclaim the performance lost due to the poor scalability of the graph partitioner, resulting in 15-46% runtime improvement. The lazy configuration results in the shortest runtime in most cases due to overlapping the application and graph partitioning computations. Our model correctly selects the best performing configuration along with the parameters, resulting in 17-46% runtime improvement.

VIII. Conclusions

We have presented a novel *lazy* approach to load balancing that decouples the load balance algorithm from the application and offloads it to achieve higher concurrency and better parallel efficiency. We have implemented a framework that allows developers to decouple their applications with a simple interface, and we have characterized the application properties and drift metrics that determine suitability of lazy load balancing. Finally, we provided a model for allocating the resources in the system based on the performance of the application and the load balance algorithm.

We have developed drift metrics to evaluate the rate of workload change in two applications: a Barnes-Hut benchmark and a production dislocation dynamics application, ParaDiS. Using our lazy load balancing approach, we have demonstrated runtime improvements of up to 46%, and we have shown that our resource allocation model can accurately predict the load balance configuration and the resource allocation that result in the lowest execution time of the application.

IX. Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-678165). The research of Amato and Pearce supported in part by NSF awards CNS-0551685, CCF-0702765, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0916053, IIS-0917266, EFRI-1240483, RI-1217991, by NIH NCI R25 CA090301-11, and by DOE awards DE-AC02-06CH11357, DE-NA0002376, and B575363. Pearce supported in part by an NSF Graduate Research Fellowship, a Department of Education Graduate Fellowship, and the Lawrence Scholar Program.

References

[1] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *Parallel and Distributed Processing Symposium (IPDPS)*, March 2007.

[2] A. Arsenlis, W. Cai, M. Tang, M. Rhee, T. Oppelstrup, G. Hommes, T. G. Pierce, and V. V. Bulatov. Enabling Strain Hardening Simulations with Dislocation Dynamics. *Modelling and Simulation in Materials Science and Engineering*, 15(6):553, 2007.

[3] J. Barnes and P. Hut. A Hierarchical O(N log N) Force-Calculation Algorithm. *Nature*, 324:446–449, December 1986.

[4] A. Bhatelé, L. V. Kalé, and S. Kumar. Dynamic Topology Aware Load Balancing Algorithms for MD Applications. In *SC'09*, November 2009.

[5] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable Line Dynamics in ParaDiS. In *SC'04*, November 2004.

[6] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-Based Barnes-Hut N-Body Algorithm. In *GPU Computing Gems Emerald Edition*, 2011.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, May 1993.

[8] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, J. Teresco, J. Faik, J. Flaherty, and L. Gervasio. New Challanges in Dynamic Load Balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, 2005.

[9] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. In *International Conference on Supercomputing (ICS)*, May 2000.

[10] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Scalable Load-Balance Measurement for SPMD Codes. In *SC'08*, November 2008.

[11] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick. Communication Avoiding and Overlapping for Numerical Linear Algebra. In *SC'12*, November 2012.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[13] R. M. Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[14] S. Kirmani and P. Raghavan. Scalable parallel graph partitioning. In *SC'13*, November 2013.

[15] M. Livny and M. Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. In *Proceedings of the Computer Network Performance Symposium*, pages 47–55, 1982.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. http://www.mpi-forum.org/docs, June 2015.

[17] O. Pearce, T. Gamblin, B. R. de Supinski, T. Arsenlis, and N. M. Amato. Load Balancing N-Body Simulations with Highly Non-Uniform Density. In *International Conference on Supercomputing (ICS)*, June 2014.

[18] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato. Quantifying the Effectiveness of Load Balance Algorithms. In *International Conference on Supercomputing (ICS)*, June 2012.

[19] J. Sancho, D. Kerbyson, and K. Barker. Efficient Offloading of Collective Communications in Large-Scale Systems. In *International Conference on Cluster Computing (ICCC)*, September 2007.

[20] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and Modeling of a Non-Blocking Checkpointing System. In *SC'12*, November 2012.

[21] K. Schloegel, G. Karypis, and V. Kumar. A Unified Algorithm for Load-Balancing Adaptive Scientific Simulations. In *SC'00*, November 2000.

[22] M. Schulz and B. R. de Supinski. P^NMPI Tools: A Whole Lot Greater Than the Sum of Their Parts. In *SC'07*, November 2007.

[23] G. Söderlind and L. Wang. Adaptive Time-Stepping and Computational Stability. *J. of Computational and Applied Math*, 185(2):225–243, 2006.

[24] K. S. Thorne. Multipole Expansions of Gravitational Radiation. *Reviews of Modern Physics*, 52:299–340, Apr. 1980.

[25] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. B. Ross, and K. Yoshii. Accelerating I/O Forwarding in IBM Blue Gene/P Systems. In *SC'10*, November 2010.

[26] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[27] J. White and J. Dongarra. Overlapping Computation and Communication for Advection on Hybrid Parallel Computers. In *Parallel Distributed Processing Symposium (IPDPS)*, May 2011.